Lenguaje Vidi. Manual de Referencia

@davidberneda v0.0.14-alpha Abril-2021

https://github.com/davidberneda/Vidi

Importante:
Borrador. TODO ES SUSCEPTIBLE DE SER CAMBIADO.

Conceptos generales

Vidi es un lenguaje de programación de tipos estrictos y orientado a objetos, que *toma prestadas* la mayoría de sus funcionalidades de otros lenguajes existentes como Java, C#, Delphi / Pascal y otros.

Por defecto no se distingue entre mayúsculas y minúsculas. Esto significa por ejemplo que `Juego` se considera equivalente a `juego`, pero opcionalmente se puede configurar para que se distinga.

Comentarios en el código

Los ejemplos en éste documento contienen *comentarios* en la misma línea de texto comenzando por: `//`

Tipos básicos de datos

Numéricos

Los números se pueden expresar de diferentes maneras:

```
123     // Entero   (Integer)
-4567   // Negativo
12.345  // Decimal  (Float)

4e2    // Exponente
-5e-3  // Exponente negativo

// Otras bases:

0xFF    // Hexadecimal base 16
0b11011 // Binaria base 2
0c217   // Octal base 8
```

Texto

Las comillas simples o dobles se usan para delimitar cadenas de texto.

```
"Hola"         // Comillas dobles
'Mundo'        // Comillas simples
"¿Cómo estás 'hoy' ?"  // Comillas simples dentro de comillas dobles
'Dijo "abc" !'  // Comillas dobles dentro de comillas simples
```

Lógicos

```
True   // Verdadero
False  // Falso
```

Otros tipos de datos

Listas o cadenas (Arrays)

```
[ 1, 2, 3 ]  // Lista simple
[ ["a","b"], ["c", "d", 'e'] ]  // Matriz (lista de listas)
```

Rangos de números enteros

Los rangos expresan valores mínimos y máximos:

```
1..10     // de 1 a 10
-12..-2   // de -12 a -2
```

Los rangos pueden usarse en diferentes situaciones, como por ejemplo al declarar una lista:

```
MiLista : Integer[1..10]  // Una lista de 10 números enteros
```

O para especificar tipos de enteros personalizados `Integer` que se beneficiarán de control de rangos (excesos):

```
// Una clase de número entero, de 1 a 3
Podium is 1..3 {}

P : Podium := 4  // <-- Error. Exceso (Overflow)

// La clase 'Podium' se puede usar también como una dimensión en una lista:

MiLista : Integer[Podium]  // equivalente a: Integer[1..3]
```

O usar un rango dentro de un bucle `for` :

`for Num in 0..1000 { }`

O en los parámetros y tipos de resultados de una función:

`MiFuncion( MiParametro : 20..1000): 4..10 { }`

Las funciones también pueden devolver rangos:

```
Meses : Range { 1..12 }
```

Un rango se puede usar también como un tipo de una lista:

```
Podiums is 1..3[10] {}  // Una lista de 10 valores enteros, cada uno de 1 a 3
```

Expresiones

Lógicas

Operadores boleanos:

`and or not xor`

Operador condicional:

`2 > 1 ? True : False   //  Ternario`

Operador de pertenencia:

```
'A' in 'ABC'  // Verdadero
5 in [1,2,3]  // Falso
```

Aritméticas

```
2 + 3 - 5 * (6 / -7) // Matemática básica

255 or 0xFF
128 and 255
64 xor 32
not 123

"Hola" + "Mundo" // Suma de textos

// Otras expresiones matemáticas usando funciones en vez de símbolos:

Math.Power(5,2)   // 5 elevado a 2 es: 25
Math.Modulo(10,3) // El resto de 10 dividido 3 es: 1

BinaryShift.Left(2,4)   // 2 << 4 es: 32
BinaryShift.Right(32768,4)  // 32768 >> 4 es: 2048
```

Comparativas

Operadores de igualdad:

`=  <>  >  <  >=  <=`

Grupos

Los paréntesis se usan para agrupar expresiones e indicar su prioridad:

`(4+2) * 6 - ((5/9) * (Abc - Xyz))`

Identificadores

Los Identificadores deben empezar por un caracter (de la `a` a la `z`) o por un guión bajo `_`, y luego opcionalmente cualquier dígito (de `0` a `9`), letra o guión bajo.

Ejemplos:

```
Abc
X123
Mi_Nombre
_Test4
_4z
```

Variables

El símbolo `:` (dos puntos) se usa para separar el identificador de una variable (el nombre) y su tipo:

Variables simples:

```
A : Integer
B : Text
```

Una variable puede opcionalmente definir un valor *por defecto* (un valor inicial) usando el símbolo `:=`:

```
F : Float := 123.45   // Inicialización del valor de la variable F
```

El tipo de una variable se puede omitir para que sea calculado en base a su valor inicial:

```
Dato ::= True      // Inferencia de tipo. (Dato es un valor lógico Boolean)
Planeta ::= Tierra   // La variable Planeta es del mismo tipo que tenga el valor Tierra
```

Las listas se declaran usando el símbolo `[]` (corchetes), y se pueden inicializar opcionalmente:

```
Colores : Text[] := [ "Rojo", "Azul" ]

Matriz : Float[ 3,3 ]   // Alternativamente: Float[3][3]
```

Los rangos y expresiones también se pueden usar para declarar las dimensiones de una lista:

```
Numeros : Integer[ 1..(2*10) ]   // 20 elementos, del 1 al 20
```

Múltiples variables del mismo tipo se pueden declarar en la misma línea de código:

```
Nombre, Apellido, Direccion : Text   // Tres variables de tipo Texto

// El valor inicial también permite múltiples variables a la vez:
X,Y,Z : Float := 1.23

// El tipo de múltiples variables se puede calcular:
A,B ::= True  // Ambas A y B son del tipo lógico Boolean
```

Asignaciones

El símbolo `:=` asigna (pone) el valor o expresión de la derecha, a la variable de la izquierda:

```
X:Text
X := 'Hola'  // <-- asignación
```

Asignaciones aritméticas (`+= -= *= /=`) :

```
Z : Integer

Z := 1

Z += 3  // Z := Z+3
Z -= 2  // Z := Z-2
Z *= 5  // Z := Z*5
Z /= 4  // Z := Z/4
```

Operador para añadir textos y listas:

```
Hola : Text
Hola := 'Hola'
Hola += ' Mundo!'  // encadenar textos

Numeros : Integer[]
Numeros += [1,2,3]  // equivale al método Numeros.Append
Numeros += 4
```

Copias y referencias

Los tipos básicos de datos (numéricos, texto y lógicos) son "*tipos por valor*". Siempre se copian cuando se asignan variables:

```
A : Integer := 123
B : Integer := A

// A y B son independientes. Modificar A no cambia B.

A := 456 // El valor de B sigue siendo 123
```

El resto de tipos (objetos, listas y funciones) siempre se asignan "*por referencia*".

```
A : Persona
B : Persona := A

// A y B apuntan a la misma variable del tipo Persona.
// Modificar cualquiera, cambia la otra:

A.Nombre := 'Juan'  // B.Nombre ahora también es Juan
```

Constantes

La palabra reservada `final` se usa para definir variables que no pueden ser modificadas una vez inicializadas (*sólo lectura*):

```
final Pi : :=  3.1415
final Hola : Text := 'Hola'

// Pi := 123  <-- Error, la constante Pi no se puede modificar
```

Para inicializar variables se permiten expresiones, incluyendo llamadas a funciones de tipos:

```
final A ::= 1
final B ::= A + 1
final C ::= Math.Square(5) // 5*5 = 25
```

Clases

Las estructuras, registros, clases e interfaces son la misma cosa en lenguaje Vidi.

```
Persona {
  Nombre : Text
}
```

Herencia de Clases

Una clase se puede heredar de otra usando la palabra reservada `is` :

```
Cliente is Persona {
  Codigo : Integer
}
```

En el ejemplo de arriba, la clase `Cliente` deriva (hereda) de la clase `Persona` .
`Persona` es la clase ancestral de `Cliente`.

Todo son clases

El módulo `sys` contiene las clases más básicas. La clase `SomeThing` es la raíz origen de todas las demás clases.

Números literales, textos, listas, etc son clases. Tipos y rutinas (funciones) son clases también. Todo es `SomeThing`.

```
SomeThing {}
```

Clases como parámetros

La palabra reservada `Self` (equivalente a *this* o *it* o *base* en otros lenguajes) representa el objeto instanciado de una clase.

```
Jose is Integer {
  Juan() {
    AlgunaClase.Prueba(Self)  // Pásamos la instancia Jose como parámetro a la
función Prueba
  }
}
```

Sub-clases y sub-métodos

Los tipos, clases y procedimientos / rutinas / métodos / funciones pueden estar anidados, ilimitados y en cualquier contexto.

```
Vida {      // clase
  Arbol {    // subclase

    Planta( Cantidad : Integer) {  // método

      Bosque is Text[] {    // subclase dentro de método
      }

      MiBosque : Bosque   // variable de la clase Planta

      SubMetodo() { }
    }
  }
}
```

Estos sub elementos se acceden usando el símbolo del punto `.` , por ejemplo para declarar una variable de un tipo sub-clase:

`Pino : Vida.Arbol`

Parámetros de Clases

Exactamente igual que en los métodos o funciones, podemos usar parámetros al declarar variables de una clase, para inicializarlas (construirlas).

```
// Parámetro: UnNombre
Cliente(UnNombre: Text) is Persona {
  Nombre:= UnNombre
}

Cliente1 : Cliente("Juan")
Cliente2 : Cliente("Ana")
```

Variables del tipo: Type

Una variable también se puede definir que sea del tipo `Type`.

```
Comida {}  // una clase simple
Fruta is Comida {}
Arroz is Comida {}

MiTipoComida : Type  // futuro: Type(Comida)
MiTipoComida := Arroz

MiComida : MiTipoComida  // <-- equivalente a MiComida: Arroz
```

Tipos Genéricos

No hace falta una sintáxis especial para definir tipos genéricos.
Los parámetros de clases de tipo `Type` se usan para especializar clases genéricas.

```
with Types

Lista(T:Type) is T[] {}    // Parámetro de tipo: Type

Numeros is Lista(Float) {}  // Lista de numéros decimales
Nombres is Lista(Text) {}   // Lista de textos
```

Conversiones de tipo (casting)

Al no existir punteros en la manera tradicional, las conversiones sólo se permiten entre tipos de clases de la misma jerarquía.

```
Clase1 {}
Clase2 is Clase1 {}

C2 : Clase2
C1 : Clase1 := C2  // Correcto, misma jerarquía

//  C2_bis : Clase2 := C1 // <-- Error, conversion debe ser explícita

C2_bis : Clase2 := Clase2(C1) // <-- Conversion correcta
```

Protección automática de conversiones de tipos

Nota: Experimental, no finalizado

```
MiClaseBase {}
MiClaseDerivada is MiClaseBase { Juan : Integer }

MiDerivada : MiClaseDerivada
MiDato : MiClaseBase := MiDerivada

// 1) Acceso a Juan prohibido, error de compilación. Conversión necesaria
MiDato.Juan := 456

// 2) Correcto, pero puede generar error en tiempo de ejecución si MiDato no es
MiClaseDerivada
MiClaseDerivada(MiDato).Juan := 789

// 3) Acceso a Juan correcto porque el "if" convierte automáticamente el tipo
if MiDato is MiClaseDerivada
   MiDato.Juan := 123  // No generará ningún error en la ejecución
```

Métodos

También llamados *rutinas*, *procedimientos* o *funciones*.

```
Area : Float { return 123 }
```

Parámetros

Los parámetros de un método se pasan por defecto como constantes de sólo lectura y no pueden ser modificados.

```
Fabrica( Ruedas : Integer ) {
 // El párametro Ruedas no se puede cambiar dentro del método.
}
```

La palabra reservada `out` delante de un parámetro significa que se debe asignar un valor obligatoriamente:

```
Piezas( Estilo:Text, out Precio:Float ):Boolean {
  Precio := 123  // <-- Se debe asignar un valor a Precio
}
```

Para devolver más de un valor ("*tuples*") se utilizan estructuras (registros):

```
Formato { Ancho:Integer Nombre:Text }  // <-- El registro

// Rutina que devuelve el registro:
MiFuncion : Formato {
  Resultado : Formato
  Resultado.Ancho := 123
  Resultado.Nombre := 'abc'

  return Resultado

  // En el futuro:  return 123, 'abc'
}

// Llamada al método para obtener el registro X:
X ::= MiFuncion
Console.Put(X.Nombre)
```

Tipos de clases anónimos

Las variables se pueden declarar junto a una declaración de clase justo después del símbolo `:` (dos puntos):

```
Planeta : { Nombre:Text, Radio:Float }
Planeta.Nombre := 'Saturno'

// Una lista se puede usar para inicializar todos los campos en orden:

OtroPlaneta: = [ 'Saturno', 58232 ]

// O una lista de listas:

Planetas : { Nombre:Text, Radio:Float } [] :=
  [
    [ 'Marte',  3389.5 ],
    [ 'Tierra', 6371.0 ]
  ]
```

Parámetros de múltiples valores

El último parámetro de un método se puede declarar con el prefijo especial `...` para permitir pasar un indeterminado número de parámetros.

En el fondo sólo es un detalle cosmético ("*syntactic sugar*") de pasar una lista de parámetros sin la necesidad de teclear los símbolos `[ ]` rodeando los valores.

```
Imprimir( Valores : Data...) {
 for Valor in Valores Console.PutLine(Valor)
```

```
}

// Ejemplos de uso:
Imprimir
Imprimir('abc')
Imprimir(123,'abc',True)

ImprimirNumeros( Numeros : Integer... ) {
 for Numero in Numeros Console.PutLine(Numero)
}

ImprimirNumeros(7,8,9,10,11)  // similar a: [7,8,9,10,11]
```

Sobrecarga de Métodos

Más de un método puede tener el mismo nombre si tiene diferentes parámetros y / o valores de retorno:

```
Escribe( Numero : Integer) {}
Escribe( Numero : Float):Integer[] {}
Escribe( Palabra : Text, Otro : Boolean) {}
```

Herencia de Métodos

Una clase derivada (hija) puede declarar métodos con exactamente el mismo nombre, parámetros y valores de retorno que su clase ancestral (padre).

La palabra reservada `Ancestor` se refiere al mismo método en la clase ancestral (padre).

```
Clase1 {
  Procedimiento() {}
}

Clase2 is Clase1 {
  Procedimiento() {
    Ancestor  // Llama a Clase1.Procedimiento
  }
}
```

Métodos no heredables (final)

Declarando métodos con la palabra reservada `final` prohibe su herencia en clases derivadas (hijas).

`final Procedimiento() {}`

Métodos Abstractos

Cuando el código de un método está vacío, se considera que es abstracto.
No hay una sintaxis especial para declararlo abstracto.

```
Prueba {
    Juan(A: Integer):Text {}  // <-- método abstracto (vacío)
}
```

Abstracto significa que no se puede llamar al método directamente (provoca un error de compilación), y que las clases derivadas deben implementar (sobreescribir) el método y llenarlo con código.

Interfaces

No hay una sintaxis especial para declarar interfaces.
Clases simples que no tengan campos (variables), y que todos sus métodos sean abstractos, se consideran interfaces.

```
MiInterfaz {
   MiMetodo( Dato : Boolean ):Text {}   // función abstracta
}
```

Las clases pueden ser derivadas de las interfaces:

```
MiClase is MiInterfaz {   // Derivada de una interfaz
  MiMetodo( Dato : Boolean ):Text { return "abc" } // Se debe implementar el
método abstracto
}
```

Interfaces ligeras (soft)

Clases que tengan métodos con exactamente el mismo nombre, parámetros y valores de retorno que los métodos de una interfaz, pueden ser usadas como instancias de esa interfaz.

```
MiClase {
  MiMetodo( Dato : Boolean ):Text { return "abc" }
}

// Este método requiere un parámetro del tipo MiInterfaz
Ejemplo( Valor : MiInterfaz) {
  Valor.MiMetodo(True)
}

Algo1 : MiClase
Ejemplo(Algo1)  // La variable Algo1 se considera del tipo MiInterfaz
```

En el código anterior, la clase `MiClase` no deriva de `MiInterfaz` pero puede ser usada como si lo fuera.

Módulos

La palabra reservada `with` importa (carga) módulos que están en archivos independientes.

Puede ser usada en cualquier parte de un archivo, no solamente al principio.
Los símbolos importados están disponibles sólo en el contexto del uso de `with`.

```
with Modulo1, Modulo2, Modulo3.MiClase

MiClase {
  with OtroModulo  // "with" en un contexto anidado

  Prueba : OtraClase // OtraClase está declarada en OtroModulo
}

// Los símbolos de OtroModulo no se pueden acceder aquí, fuera del contexto de
MiClase
```

Los nombres de archivo de los módulos pueden contener espacios o caracteres que no están permitidos en identificadores. En éste caso, la sintaxis de `with` permite entrecomillar el nombre del módulo como si fuera un texto literal:

```
with "Mi módulo con espacios"
```

Un nombre "alias" se puede usar para cambiar el nombre de un módulo. Por ejemplo para reducir su longitud o para evitar conflictos de duplicación.

```
with Juan:= Mi_Modulo_Largo.Mi_Clase   // Juan es un alias

Juan1 : Juan  // variable del tipo Mi_Clase
```

Grupos de módulos

Un módulo se puede usar como un grupo que agrega otros módulos en un sitio único central:

```
// Uso de módulos con nombres alias
with M1:=Modulo1, M2:=Modulo2  // etc

// Declaración de los módulos como nuevas clases
Modulo1 is M1 {}
Modulo2 is M2 {}
```

Al usar el módulo anterior, el contenido de `Module1` y `Module2` estará disponible directamente sin la necesidad de añadir los `with` correspondientes.

Visibilidad de Elementos

La palabra reservada `hidden` prefija una clase, variable o método haciéndolo invisible fuera de su contexto.

```
hidden MiClase {
  hidden MiVariable : Integer
  hidden MiFuncion : Boolean {}
  hidden MiSubClase {}
}
```

Elementos ocultos que no se usan producen un error en tiempo de compilación.

Elementos compartidos a nivel de tipo

La palabra reservada `shared` significa que un elemento (variable o método) pertenece al nivel de su tipo, no al nivel de las instancias del tipo.
Esto es equivalente a las "*class variables*" en otros lenguajes.

```
Colores {
  shared Defecto : Text := "Rojo"
}

Colores.Defecto := "Azul"   // Uso a nivel de tipo, sin ninguna instancia
```

Los métodos a nivel de tipo se señalan automáticamente sin necesidad de prefijarlos. No hay una sintaxis especial para declararlos.
Cuando un método no accede a ningún elemento que no sea de nivel de tipo, se considera compartido (`shared`).

```
Colores {
  shared Defecto : Text := "Rojo"

    // Método a nivel de tipo, no hace falta la palabra reservada: shared
  PonerDefecto( Valor : Text) { Defecto := Valor }
}

Colores.PonerDefecto( "Green" )
```

Espacio de Nombres

No hay una sintaxis especial para declarar espacios de nombres.
Clases con ninguna variable ni métodos se consideran espacios de nombres.

```
// Modulo1
MiEspacio {
  MiClase {}
}
```

Módulos con espacios de nombres duplicados se pueden mezclar para agregar (contribuir) nuevas clases a esos espacios de nombres:

```
// Modulo2
MiEspacio {
  OtraClase {}
}
```

La palabra reservada `with` se puede usar también para referenciar a sólo un sub elemento en vez de a todo el contenido de un módulo:

```
// Modulo3
with Modulo1.MiEspacio,
     Modulo2.MiEspacio

Prueba is OtraClase {
  Algo : MiClase
}
```

Tipos estrictos

Derivar un tipo de otro es una forma práctica de control estricto de tipos:

```
// Alias de tipos

Dia is Integer {}
D : Dia

Mes is Integer {}
M : Mes

D := M  // <-- Error. Tipos diferentes.

Clase1 {}
Clase2 is Clase1 {}

C1 : Clase1
C2 : Clase2
// ERROR: C2 := C1  <-- equivalente pero prohibido (control estricto)
```

Introspección de Tipos (reflection)

La clase `Type` incluye métodos para inspeccionar tipos existentes:

```
// Comprobación de tipo:
if Type.is( C1, Class1 ) ...

// Obtener la lista de métodos de un tipo o instancia:
Methods: Method[] := Type.Methods( C1 )
```

Extensiones

Los tipos se pueden ampliar (extender) con nuevos métodos y subclases, en cualquier contexto incluso en módulos diferentes.
Por ejemplo podemos declarar una clase en un módulo:

```
// Módulo 1
MiClase {
}
```

Y entonces, en el mismo módulo o en otros módulos, podemos declarar nuevos métodos y subclasses de `MiClase` :

```
// Módulo 2
with Modulo1

MiClase.MiProcedimiento() {}   // Nuevo método extendido
MiClase.MiSubClase { X:Float }  // Nueva subclase extendida
```

Estos nuevos elementos extendidos se pueden usar de manera normal, también en otros módulos.

```
// Módulo 3
with Modulo1, Modulo2

Juan : MiClase
Juan.MiProcedimiento()   // Llamada a una extensión como si fuera un método
normal

Ana : MiClase.MiSubClase
Ana.X := 123
```

Las extensiones, como cualquier tipo normal, sólo se pueden acceder desde el contexto en el que han sido declarados.

Los tipos extendidos pueden también ser extendidos:

```
MiClase.MiSubClase.MiNuevoMetodo() {}
Ana.MiNuevoMetodo()
```

Tipos de Funciones

Un tipo se puede usar como una declaración de una función:

```
MiTipoFuncion is (A:Text, B:Integer):Float {}
```

Este tipo puede luego ser usado como un tipo normal:

```
Juan(Funcion: MiTipoFuncion) { Funcion('Hola',123) }
```

Para ser compatibles con `MiTipoFuncion`, las funciones deben tener mismos parámetros y retorno:

```
MiFuncion(A:Text, B:Integer):Float {
  Console.PutLine(A, ' ', B.AsText)
}
```

La función `MiFuncion` se puede llamar o pasar como parámetro.

```
Juan(MiFuncion)  // muestra 'Hola 123'
```

Funciones Anónimas

También llamadas *lambdas* o *callbacks* en otros lenguajes, son funciones que se pueden pasar como parámetro sin previa declaración (de manera "*inline*
") ni nombre:

```
// Igual que en el ejemplo anterior "MiFuncion", pero de forma anónima, sin
nombre
Juan(
 (A:Text, B:Integer):Float { Console.PutLine(A,' ',B) }
)
```

O se pueden asignar a variables del tipo de la función:

```
// Como en el ejemplo anterior, pero usando una variable
MiVariable_Funcion : MiTipoFuncion := { Console.PutLine(A, ' ', B) }

Juan(MiVariable_Funcion)
```

Tipos Enumerados

La sintaxis `is {}` se usa para declarar enumeraciones.

`Colores is { Rojo, Verde, Azul, Amarillo }`

Los tipos enumerados se pueden usar en variables y constantes:

`MiColor : := Colores.Azul`

También se pueden usar como dimensiones de listas:

```
Nombres : Text[Colores]  // Lista de cuatro elementos de tipo Text
Nombres[Colores.Verde] := "Me Gusta El Verde"
```

Y en las instrucciones `for in` para iterar todos los elementos de una enumeración:

```
for Color in Colores {
  Console.PutLine(Color)
}
```

Instrucciones

Asignación

```
a := b
b := c + d
```

If (si condicional)

```
if a=b
   Hola
else
   Adios
```

While (mientras)

```
while a>b {
  if a=0
     break   // sale del bucle "while"
  else
     a:= a - 1
}
```

Repeat (repetir)

```
repeat {
  b += 1

  if b=5
     continue   // "continue" salta al inicio de "repeat"

} until a<>b

// Instrucciones en una sola linea de código no necesitan { }

repeat
  b +=1
until b>5
```

For (bucle)

Un bucle simple sin ningún contador:

```
for 1 to 10 {}  // diez veces
for 5..7 {}   // tres veces
```

Un rango de números enteros:

```
for t in 1..10 {}   // diez veces
```

Bucle tradicional usando la palabra reservada `to` :

```
a ::= 5    b ::= 7
for x: := a to b {}   // tres veces
for y: := 1+a to 9 {} // cuatro veces, de 6 a 9
```

La variable opcional (el contador):

- No se puede reutilizar ni acceder fuera del bloque de código de `for`
- No se puede modificar dentro del bucle
- No puede ser una variable que ya exista en el contexto
- Su tipo siempre se calcula automáticamente (inferencia)

La palabra reservada `in` se usa en bucles con tipos enumerados:

`for c in Colores {}`

Y también para iterar los elementos de una lista:

```
Numeros: := [ 6,2,9 ]
for i in Numeros { Console.Put(i) }    // iteración de una lista
```

La lista se puede declarar en línea sin usar una variable:

```
for i in [ 6,2,9 ] // el tipo del contador "i" se infiere automáticamente
```

Una expresión del tipo texto `Text` es una lista de caracteres que puede ser iterada:

```
for c in "abc" {} // para cada caracter
```

When (cuando)

También llamado *switch*, *select* o *case* en otros lenguajes.

```
Nombre: := "Juan"

when Nombre {
  "Juan"  { HazEsto }
  "Maria" { HazOtro }
else
   Ejemplo
}
```

Soporta expresiones comparativas:

```
numero ::= 5
abc ::= 3

when abc+numero {
  < 3 { Console.PutLine('Menor de 3') }
    4 { Console.PutLine('Igual a 4') }
 <> 6 { Console.PutLine('Diferente de 6') }
else { } // en caso contrario
}
```

La ejecución termina cuando se encuentra la primera condición cierta.

Return (retornar)

La instrucción `return` termina un método, devolviendo opcionalmente un valor si el método es una función.

```
Prueba {
  Juan() { return }
  Maria:Text { return "abc" }
}

// La palabra return es opcional si es la última expresión de la función:
Square(X:Float) { X*X }
```

Try Catch (intento y captura)

El tratamiento de los errores (excepciones) sigue el estándar de otros lenguajes usando las palabras reservadas `try`, `catch` y `finally` .

El código dentro de un bloque `try` está protegido de manera que si sucede un error, el bloque de código `finally` siempre se ejecuta:

```
try {
  X::=1/0   // <-- error división por cero !
}
finally {
  Console.PutLine('Siempre se ejecuta')
}
```

El bloque de código `catch` se ejecuta cuando sucede un error dentro del bloque `try` :

```
try { X::=1/0 }
catch {
  // Código opcional, puede estar vacío {}
  Console.PutLine('Ha sucedido un error')
}
```

La palabra reservada `catch` puede opcionalmente especificar un tipo (la clase `MiError` en éste ejemplo), para capturar solamente éste tipo de error:

```
MiError { Codigo:Integer }
Juan() { Exception.Raise(MiError) }  // <-- un ejemplo de generación del error

try { Juan() }
catch MiError {
  Console.PutLine('Ha sucedido MiError')
}
```

Si el tipo de error en `catch` se prefija con un nombre de variable (`X` en el ejemplo siguiente), tenemos acceso a los campos del tipo del error:

```
try { Juan() }
catch X:MiError {
  Console.PutLine('Ha sucedido MiError: ', X.Codigo)
}
```

Los bloques `catch` y `finally` pueden coexistir (`finally` debe estar después de `catch` por claridad):

```
try { Juan() }
catch { Console.PutLine('Ha sucedido un error') }
finally { Console.PutLine('Siempre se ejecuta') }
```

Múltiples bloques `catch` se pueden usar para responder a tipos de error diferentes. No se permiten tipos duplicados.

```
try { Juan() }
catch X:MiError { Console.PutLine('Ha sucedido MiError: ', X.Codigo) }
catch DivideByZero { Console.PutLine('División por cero') }
// catch DivideByZero {} <-- error de compilación, tipo duplicado
```

Recursividad

Los métodos pueden llamarse a sí mismos de manera recursiva:

```
Factorial(x:Integer):Float {
  x=0 ? 1 : x * Factorial(x-1)
}

Factorial(5)  // Devuelve 120
```

Declaraciones diferidas

Hay situaciones en donde se debería llamar a métodos que todavía no están declarados. Estos casos se tratan automáticamente, sin ningún tipo de sintaxis especial. (Nota: En construcción)

```
Prueba(Trabaja:Boolean) {} // <-- método vacío, borrador
```

```
Llama() {
  Prueba(False)  // <-- Prueba aún no está declarado
}

// Aquí se reemplaza el borrador:
Prueba(Trabaja:Boolean) {
  if Trabaja
     Llama
}

{
  Prueba(True)
}
```

Propiedades

No hay una sintaxis especial para declarar propiedades.
Una propiedad "*getter*" puede ser un campo o variable:

`Juan : Integer := 123`

O una función:

`Juan : Integer { return MiValor }`

Y opcionalmente, el escritor de la propiedad "*setter*" que es sólo una función con el mismo nombre y un parámetro:

`Juan(Valor:Integer) { MiValor:=Valor } // escritor`

El compilador ajusta el acceso a la propiedad transparentemente:

`Juan:=123  // aquí se llama al método escritor: Juan(123)`

Finalizadores

Las clases pueden definir un método sin nombre, sin parámetros, prefijado con `final` que se llama cuando las instancias de esa clase terminan su contexto (se destruyen).

```
Tienda {
  Console.PutLine( 'Abierto !' )

  final {
     Console.PutLine( 'Cerrado !' )
   }
}

{
  MiTienda : Tienda
  // hacemos algo con MiTienda ...

} // <-- aquí al final del contexto, se llama al finalizador de MiTienda
```

Operadores de Expresión

Nota: Experimental, en construcción

Nuevos operadores de expresiones se pueden implementar para añadir sintaxis cosmética ("*syntax sugar*") usando símbolos o palabras reservadas.

```
// Declara un nuevo operador "is", usando la función ya existente Type.is
Operator.is := Type.is

// Ejemplo de uso
Juan : Boolean

// Expresiones equivalentes
if Type.is(Juan, Boolean) Console.PutLine('Ok')

// Usando el nuevo operador
if Juan is Boolean Console.PutLine('Ok')

// Los operadores básicos existentes como +, -, *, >, < etc podrían ser
teóricamente re-implementados como extensiones.
// El compilador busca el método más adecuado de acuerdo a los tipos de la
izquierda y de la derecha de la expresión.

A + B   // Llama a Integer.Add(A,B) si ambos A y B son compatibles con el tipo
Integer
```

Sintaxis

Palabras reservadas

```
ancestor
and
break
catch
continue
else
False
final
finally
for
hidden
if
in
indexed
is
not
or
out
repeat
return
```

```
self
shared
to
True
try
until
when
while
xor
with
```

Símbolos reservados

```
{ }   // bloque de código
.     // membresía  Juan.Metodo
[ ]   // listas   [1,2,3]
:=    // asignación Juan:=123
:     // declaración de tipo Juan:Integer
,     // parámetros 1,2,3
( )   // grupos de expresiones, parámetros
..    // rangos 1..100
...   // múltiples parámetros
?     // expresión condicional A=B ? 1 : 2
>     // mayor que
>=    // mayor o igual que
<     // menor que
<=    // menor o igual que
=     // igual
<>    // diferente
+     // suma
-     // resta
*     // multiplicación
/     // división
```

Comentarios

```
// Comentarios en la misma línea

/*
  Múltiples lineas de
  comentarios
*/

Ejemplo : Text := "permite" + /* comentarios */ "alrededor del código"
```